import RemoteCodeBlock from "@site/src/components/RemoteCodeBlock"; import Admonition from "@theme/Admonition";

# Breakdown: simple-blind-arbitrage

Our goal is to make an *arbitrage bot* that uses flash loans to fund its trades. Arbitrage opportunities arise when one user makes a trade that results in a significant price shift on one exchange. If there is a trading pair on another exchange for the same tokens, and the difference in price between the two exchanges is large enough, we can turn a profit by sending a two trades immediately after the one that caused the price shift: one to buy tokens on one exchange, one to sell them on another. Sending one or more transactions immediately after another is known as **backrunning**.

The goal of backrunning is to guarantee that our trades are first in line to execute after the result of some transaction (e.g. a price shift). This allows us to calculate the optimal trade at execution-time with certainty that it will either execute the ideal arbitrage, or revert.

> This guide is based off of simple-blind-arbitrage. Before you continue with this guide, we recommend skimming the README for a technical overview of the system. If this raises more questions than it answers, that's OK! This guide will break down each component of the bot in detail.

Before we backrun any transactions, we need to create a smart contract to make trades for us. The backrun transaction we send will call a function on the contract that executes the arbitrage. So how does the smart contract execute an arbitrage?

To find out, let's examine this bot's strategy: *onchain searching*.

## Onchain Searching

MEV-Share introduces some key differences from more common strategies that you may have seen elsewhere (e.g. simple-arbitrage, subway, rusty-sando). These bots rely on *full transaction simulations* (with signatures) to calculate the optimal trade *off-chain*. Calculating trades off-chain saves gas, but comes at the cost of uncertainty. Based on your bundle's placement in the block, the state your transaction relies on may change, which could potentially invalidate your transaction.

On MEV-Share, pending transactions typically expose *less data* than transactions in the public mempool. Transaction signatures are *always* hidden from searchers. Simulation-based strategies (e.g. rusty-sando) on these transactions are usually not possible, since the amount traded by the user is typically hidden. That being said, users can choose to reveal more data to searchers, so all the classic strategies can still be used; they'll just land less often.

The strategy we'll use is called "onchain" searching: we calculate how much to trade within the "trade" itself, effectively executing the searching strategy & algorithm *on the blockchain* ("onchain"). The advantage of this strategy is that it has direct access to onchain state, meaning it can *always* calculate the most optimal arbitrage trade parameters, unlike off-chain strategies.

We send a backrun for every transaction that touches the tokens we're interested in trading, and rely on Flashbots to prevent unprofitable trades from landing on-chain. The *amount* we buy & sell in our arbitrage trades is derived from the prices of the assets on the blockchain at the time of execution. Because we place our transaction behind another user's transaction ("backrunning"), the price that our transaction sees is the price which has been changed by the user's trade. This is where we get our arbitrage opportunity.

## Arbitrage Contract

We'll start by looking at a ready-made smart contract, and then break it down piece by piece.

This is the **core logic contract**, which contains functions for performing arbitrage between Uniswap-V2-like exchanges (e.g. UniV2 / Sushiswap). Later on, we'll create other contracts that inherit this one, so that we can add custom asset management logic (flash loans, where to store profits, etc.) without having to rewrite all the Uniswap-centric logic, which you likely won't need to change.

This may look complicated, but by the end we'll have explained every line of code. We'll start at the top with Imports & Interfaces.

### Imports & Interfaces

We start by importing some contract interfaces `openzeppelin/access/Ownable.sol` and `./IWETH.sol`. Ownable allows us to restrict certain functions to the contract owner. IWETH allows us to deposit/withdraw ETH for WETH. We need WETH because Uniswap (V2/V3) only supports ERC20 tokens.

We also define a couple interfaces ourselves: `IUniswapV2Pair` and `IPairReserves`. We could import these from the official Uniswap contract library like we did with OpenZeppelin for the Ownable contract, but that comes with a lot of bloat for our project. In this case, we only need four functions from `IUniswapV2Pair`, and the struct definition of `PairReserves` from `IPairReserves`.

Defining these interfaces allows us to interact with other smart contracts directly, as we'll see in the next sections.

## Abstract Contract

It's important to remember that this is an abstract contract, meaning that to use it, we'll need to write another smart contract that extends it (using the `is` keyword). This contract is responsible for implementing the capital-management strategy; where to keep money, where/how to get it; as well as any other custom logic such as fee payments, etc. We'll do a walkthrough of a finished implementation with flash loans after we break down the core logic contract. Read on to learn how our arbitrage algorithm works.

## Calculating the Optimal Arbitrage

To illustrate what the algorithm does, we plot profit (in ETH) from an arbitrage, where we buy `amount_in` tokens for WETH on one exchange and sell them all for WETH on another.

As you can see, the optimal amount_in to buy is approximately 35 ETH, but that's just eyeballing. How do we calculate the exact optimal point?

The following function calculates the optimal trade amount such that gross profit is as high as possible:

- let $F$ = `FEE` = 997
- let $D$ = `FEE_DIVISOR` = 1000
- let ${R_i}_{A|B}$ = `reserveIn` for exchange A or B
    - this refers to the reserves of the token that we're paying *into* the trade
- let ${R_o}_{A|B}$ = `reserveOut` for exchange A or B
    - this refers to the reserves of the token that we're getting *out* of the trade

$$ profit_{gross} = { \left(\sqrt{F^2 \cdot {R_o}_A \cdot {R_o}_B \over {R_i}_B \cdot {R_i}_A} - D\right) \cdot {R_i}_B \cdot {R_i}_A \cdot D \over \left(F \cdot {R_i}_B \cdot D \right) + \left( F^2 \cdot {R_o}_A \right) } $$

*For example, if we're arbitraging WETH -> TKN on exchange A, then TKN -> WETH on exchange B, our variables would be:*

- ${R_i}_A$ = `WETH.reserves`
- ${R_o}_A$ = `TKN.reserves`
- ${R_i}_B$ = `TKN.reserves`
- ${R_o}_B$ = `WETH.reserves`

How to derive this formula is beyond the scope of this document, but if you want to dig deeper, check out this paper.

This formula is implemented by the `getAmountIn` function in our smart contract, which relies on `getNumerator` and `getDenominator` to do the math (and to avoid "stack too deep" errors).

Now that we know how to calculate the optimal amount of WETH to send for an arbitrage, let's put it to use.

### _executeArbitrage

`_executeArbitrage` is the core function responsible for looking up trading prices, calculating the optimal buy/sell amounts, and executing the two trades that make up the arbitrage. It only takes three arguments:

```
solidity function _executeArbitrage( address firstPairAddress, address secondPairAddress, uint percentageToPayToCoinbase ) ...
```

We just tell it which token pairs to trade, and how much profit to tip the validator. We choose to make `percentageToPayToCoinbase` a function argument because as competition increases, you may have to pay more to the validator to be selected over another competing bundle. This may change frequently, so it's important to monitor the bot and adjust the tip as needed.

The function starts by reading the smart contract's own WETH balance. This is used later to verify our profits. We use the pair addresses to instantiate uniswap Pair contracts, which we pass to `getPairData` to read the reserves, which we then use to calculate the optimal arbitrage with `getAmountIn(firstPairData, secondPairData)`.

Uniswap token pairs refer to their tokens as `token0` and `token1`; `token0` being the one whose address is numerically less than the other (e.g. 0x0123 < 0x0234); so we need to discern which token of the pair's two tokens is WETH. Our `getPairData` function sets this in the `isWETHZero` field. If WETH is token0, then we'll trade token0 -> token1 on exchange A, then token1 -> token0 on exchange B. If WETH is token1, then we just switch "token0" with "token1" and apply the same formula.

Once we know which token is which, we can make assertions about our profits and calculate how much to send for the second trade. To calculate how many tokens we'll receive from a single trade, we use a custom `getAmountOut` function. It's adapted from the UniswapV2 Library contract; we just removed the safety checks to save gas. We don't need guard rails since Flashbots will prevent reverting transactions from landing onchain.

We use the values calculated from `getAmountOut` as inputs to the token pairs' swap functions: we use `amountIn` (the "optimal arbitrage" value) as the amount (of WETH) to send for the first trade, then use the output (`firstPairAmountOut`) of the first trade as the input to the second trade, selling all the tokens we bought from the first exchange to the other.

Once we've executed our trades, we should expect to have more ETH (or WETH) than we started with. But that won't always be the case. To ensure that we don't pay for an unprofitable trade, we check the WETH balance at the end of the `_executeArbitrage` function. If the balance isn't greater than when we first called the function, the transaction will revert. This protects us from malicious tokens, unforeseen market conditions, and a variety of other ways you can lose your money.

When we do turn a profit, we need to pay some of it to the validators/builders in order to get our transactions on-chain. Block builders have differing preferences & ordering algorithms, but a good rule of thumb is to use `maxBaseFeePerGas` and `maxPriorityFeePerGas` values that are slightly higher than the market average, and then tip a percentage of your profits to the builder. MEV-Share uses this builder tip to pay the user; this is the "MEV kickback."

## Fee Considerations

If you have no competition, you only need to send as much as is required to pay the user's kickback, which includes the gas cost of the payment transaction and a non-zero kickback.

To prevent negligible-value tips from being sent, MEV-Share uses 30,000 gas to represent the cost of the user payment transaction, instead of 21,000. The minimum tip payment required for a MEV-Share bundle to be eligible for inclusion is as follows:

- let $F_B$ = `block.baseFee`

$$ T_{min} = 30000 \cdot F_B $$

This represents the **total surplus ETH** that the coinbase must have received *in addition to gas fees* for all transactions in the bundle. Note that we don't include a priority fee for the tip; this is because the builder will send the transaction that pays the user with a priority fee of 0.

Bots that only send two-transaction bundles will probably want send this tip in the backrun tx. However, this may also be paid by multiple transactions. Also note: sending your tip via `block.coinbase.transfer` is not technically necessary; it is possible to achieve the same effect by simply increasing the gas price (`maxBaseFeePerGas` and `maxPriorityFeePerGas`) of your backrun transaction(s).

Tipping the minimum is unlikely to result in your bundle landing on-chain unless you have zero competition. If there are other searchers competing to include the same transaction in their bundles, you will have to pay a higher percentage of your profits to outbid them.

You may try tipping low to start, but over time, you should expect competition to increase. With healthy competition, your tip is likely to be around 80-90%.

- let $P_{net}$ = net profit (profit after paying gas fees)
- let $\gamma$ ($\gamma \in \R; \gamma > 0, \gamma < 1$) = percentage of profit sufficient to outbid competition

$$ T_{optimal} = (30000 \cdot F_B) + (P_{net} \cdot \gamma) $$

How you find the exact optimal value of $\gamma$ is a matter of trial-and-error, and will continually change, but you'll probably find success between 50-90%.

By default, 90% of the tip $T$ is sent to the user whose transaction was included by your bundle. This can be changed with the `validity` parameter in the `mev_sendBundle` params, but it suffices for now to keep the default settings and just send less from the smart contract.

## Compile & Deploy (optional)

If you want to run a capital-intensive strategy (not using flash loans) you'll have to deploy your own contract. This is not required if you use our flash loan contract. The flash loan contract is designed to send profits to the caller when the arbitrage is done. This allows anyone to execute arbitrages without paying to deploy the contract. The tradeoff with this is that it costs more gas to transfer the profit to your wallet than it does to keep the money in the contract. However, if you decide to deploy your own contract, its transactions will use less gas (at the risk of your contract containing a bug that might compromise the funds), but will only land bundles if it holds enough capital to buy the required tokens.

A simple way to deploy contracts is to use **forge** from Foundry:

```bash
```

## compile contracts

```
forge build
```

## to test with a local fork:

```
anvil -f $MAINNET_RPC_URL --chain-id 1 &
```

## these vars are set to deploy on local fork; change as/if needed

```
export PRIVATE_KEY="0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80" export WETH_ADDRESS="0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2" export RPC_URL="http://localhost:8545"
```

## deploy flashloan arb contract

```
forge create -r $RPC_URL --private-key $PRIVATE_KEY BlindBackrunFlashLoan --constructor-args $WETH_ADDRESS
```

## or deploy capital-intensive contract

```
forge create -r $RPC_URL --private-key $PRIVATE_KEY BlindBackrun --constructor-args $WETH_ADDRESS ```
```

*output:*

```
No files changed, compilation skipped Deployer: 0xf39Fd6e51aad88F6F4ce6aB8827279cffFb92266 Deployed to: 0xc075BC0f734EFE6ceD866324fc2A9DBe1065CBB1 Transaction hash: 0xe9567cce60dfdc1f815f4724340228f2af77ee5cc157a69d07c4b270fcab3a30
```

### Tradeoffs

Our onchain searching strategy is relatively straightforward, but it has its drawbacks:

- calculating trade amounts onchain costs gas, making the strategy less efficient
- only Uniswap-V2-like trading pairs are compatible with this strategy
    - Uniswap V3 uses a different algorithm, which is very costly to compute onchain

#### Offchain Searching

The more data you can compute offchain, the less gas you have to spend. This gives you more ETH to tip with; better profit margins or a competitive edge.

However, off-chain data is not always as accurate as on-chain data. This is because other transactions may affect the state of the blockchain (e.g. the price of a trading pair) if they are placed before your bundle in the block. The very first position in an upcoming block ("top of the block") is the only one that's guaranteed to match the state of the last block.

Unless you're the block builder (or have privileged access to their orderflow), you have no way of knowing where in the block your transaction will be placed, so you have no way of knowing whether the off-chain state used to calculate your strategy's parameters (e.g. your trade amounts) is still accurate. You may add logic in your smart contract that reverts if the transaction isn't placed in the first position, which will guarantee that your information is accurate, but may lower your chances of being included, depending on the builder's transaction-sorting algorithm.

#### Other Exchanges

We strictly use Uniswap V2 in this guide/bot because its pricing algorithm is simple, making arbitrages easily calculable. Uniswap V2 and Sushiswap use the same pricing algorithms, so we efficiently arb between those two exchanges. Uniswap V3 math is more complicated, making arbitrages on V3 very inefficient to calculate onchain.

However, Uniswap V3+ processes much more trade volume than V2. To improve your profits, consider developing a strategy that integrates Uniswap V3 into your own contract. It will likely involve probabilistic methods. Also note: Uniswap V4 uses the same pricing math as V3.

MEV-Share also shares hints for swaps on Balancer and Curve, from users with the default hint settings.

---

Now that the core contract is ready, let's add flash loans. Read on in the next page.